Project Spitfire - Towards Grid Web Service Databases

Informational Document
Global Grid Forum 5
Edinburgh, Scotland, July 21-24, 2002

William H. Bell[1], Diana Bosio[2], Wolfgang Hoschek[2], Peter Kunszt[2], Gavin McCance[1], and Mika Silander[3]

[1] Department of Physics and Astronomy, University of Glasgow, Scotland
[2] CERN IT Division - European Organization for Nuclear Research, 1211 Geneva 23, Switzerland
[3] Helsinki Institute of Physics, Helsinki, Finland

Abstract. Many Data Grid services maintain persistent metadata in remote relational databases. However, existing relational database systems are neither grid enabled nor web service enabled, adversely affecting cross-organizational interoperability and reuse. The European Data Grid's Data Management Work Package addresses these issues with *Spitfire*. The Spitfire service grid-enables a wide range of relational database systems by introducing a uniform service interface, data model, network protocol and security model. These are based on widely accepted standards and neutral with respect to programming language, platform and database product. In this paper we briefly describe the latest stable Spitfire release. We then discuss in detail work in progress towards a web service based architecture, its security model and client APIs, illustrated by practical examples.

1 Introduction

Grid technology attempts to support flexible, secure, and coordinated information sharing among dynamic collections of individuals, institutions and resources. This includes data sharing but also includes access to computers, software and devices required by computation and data-rich collaborative problem solving [1]. These and other advances of distributed computing are necessary to increasingly make it possible to join loosely coupled people and resources from multiple organizations.

For example, the international High Energy Physics (HEP) research community is facing a substantial challenge in joining a massive set of loosely coupled people and resources from multiple distributed organizations. The HEP community comprises thousands of researchers globally spread over hundreds of laboratories and university departments. These include CERN (International); Stanford, Berkeley, Caltech, Fermilab (USA); INFN (Italy); RAL (UK); IN2P3 (France), and more organizations from some 80 countries such as Austria, Brazil, China, Greece, India, Japan, Pakistan, Mexico, Portugal, Russia, Spain, etc. What is unique is that these dispersed researchers *jointly* undertake mammoth projects; these projects often have a lifetime of more than ten years, and include the use distributed resources such as thousands of network services, tens of thousands of CPUs, WAN Gigabit networking as well as Petabytes of disk and tape storage [2]. Thus, the European DataGrid (EDG) [3, 4] is under construction. In DataGrids, many small and large services spread over multiple loosely coupled organizations work together to provide access to and management of massive amounts

of distributed data [5]. Services such as a replica location service [6], hyper registry [7], job scheduler [8] and performance monitor [9] maintain persistent metadata in databases.

With mature open source and commercial implementations available from the low to high end, relational database management systems (RDBMS) are established as the reliable, efficient and secure storage backbone of IT industry. For example, MySQL, Oracle, IBM DB/2 and Postgres are in wide spread use. The JDBC API [10] defines a uniform vendor independent way to communicate with a wide range of RDBMSs. It is a thoroughly accepted industry standard and optimized drivers exists for every serious RDBMS. However, JDBC is Java specific, and it does not specify a standard network protocol.

Existing RDBMSs are neither grid enabled nor web service enabled. No Grid standards exist for uniform database interfaces, data model, network protocols and security, adversely affecting cross-organizational interoperability and reuse. In addition, the complexity of RDBMSs is often prohibitive for commodity usage. Until these DataGrid requirements are addressed more thoroughly, applications will continue to use dozens of varying proprietary approaches towards metadata storage and retrieval.

EDG's Data Management Work Package (http://cern.ch/grid-data-management) addresses these issues with *Spitfire* (http://cern.ch/hep-proj-spitfire), which provides a uniform way to access many RDBMSs through standard Grid protocols and well-published Grid interfaces. The Spitfire middleware is inserted into the control and data path between client and RDBMS. Spitfire grid-enables a wide range of relational databases by introducing a uniform service interface, data model, network protocol and security model. These are based on widely accepted standards (e.g. OGSA [11], WSDA [12], URIs [13], HTTP [14], TLS [15], GSI [16], XML [17], SOAP [18], WSDL [19]) and neutral with respect to programming language, platform and database product. Spitfire is easy to use, efficient, flexible, and some projects (Alice, Atlas) are starting to use it.

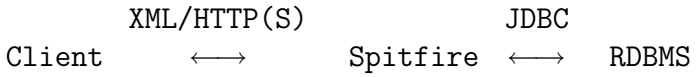
This paper is organized as follows. Section 2 describes the architecture of the latest stable Spitfire release (v1.1.0). This release is currently under active use by the EDG application work packages on the experimental testbeds, and support will continue, although development will be frozen other than for bug fixes. Work now concentrates on evolving Spitfire towards a Grid web service (v1.2.0alpha), described in the remaining sections. Section 3 describes the architecture of the web service Spitfire, whereas section 4 outlines the security mechanisms for authentication and authorization. In section 5 an overview of the client API is presented. Section 6 describes the deployment by means of examples. Section 7 outlines interesting directions for future development. Section 8 concludes this paper. Finally, appendices A and B describe in detail the complete API and extended example deployments, respectively.

2 Current Spitfire Architecture

In this section we describe the architecture, data model, network protocol and security model of the latest stable Spitfire release (v1.1.0), also dubbed *XSQL Spitfire*.

Overall Architecture. The Spitfire middleware is placed between client and RDBMS. The Spitfire server, the client, and RDBMS can run on the same machine but can also run on three different machines separated by the LAN or WAN. In the latter case, the security of

the connection between the client and the server can be assured via the use of HTTPS, whereas the security of the connection between the server and the RDBMS relies entirely on the JDBC driver used. the The Spitfire server design allows the server to be called from remote command line tools (jwget, wget, curl et al) or custom client applications, for example using the xerces XML parser (Java, C/C++, Perl) and httpclient (Java), libwww (C/C++), libwww-perl (Perl) or httplib (Python). In addition to these client applications a browser client may be used to access the Spitfire server. When contacted by a browser client the Spitfire server returns Resultsets as formatted HTML tables instead of XML if indicated in the request statement. The Spitfire server is implemented as a Java servlet running within a *container* of a virtual hosting environment such as the Apache Tomcat servlet container [20]. The architecture can be summarized as follows:



Data Model. Widespread availability of efficient parsers, available for virtually every programming language, has allowed XML to become an established industry standard to represent flexible interoperable data. The Spitfire client and server communicate relational data in canonical XML format. Canonical XML defines a 1:1 mapping from relational tables to XML format and vice versa: a relational table corresponds to a XML ROWSET element, a row corresponds to a nested XML ROW element, and a column is mapped to a nested element with the same name, filled with the value of the row's column. Let us introduce canonical XML by example. Assume we have a relational table with Logical File Name (LFN) and Physical File Name (PFN) columns:

LFN	**PFN**
lfn://cms.org/file1	ftp://host1.cern.ch/myfile1
lfn://whatever.org/file2	ftp://host3.anl.gov/data/file.tar.gz

The corresponding canonical XML representation of the table looks as follows:

```
<ROWSET>
       <ROW>
             <lfn> lfn://cms.org/file1 </lfn>
             <pfn> ftp://host1.cern.ch/myfile1 </pfn>
       </ROW>
       <ROW>
             <lfn> lfn://whatever.org/file2 </lfn>
             <pfn> ftp://host3.anl.gov/data/file.tar.gz </pfn>
       </ROW>
</ROWSET>
```

Network Protocol. With HTTP command line clients, GUI browsers, client programming language APIs and server frameworks available for virtually every language and architecture, HTTP and its SSL based secure variant HTTPS have been firmly established as the industry standard protocol for interoperable networking (and not just as vehicle to deliver HTML). Hence, Spitfire is designed so that clients can read and write over HTTP(S) into any RDBMS. Our HTTPS configuration is fully GSI enabled [16] and compatible by using *CoG*, the Java port of the Globus toolkit developed at Argonne National Laboratories.

Service Interface. Spitfire serves as an effective way to shield clients from any backend implementation details. In effect, it provides an abstract function invocation interface based on URL encoding [13] of function names and parameters for SQL standard functions such as query, insert, delete, and update, as well as encoding of custom functions. For example, to get all physical files names for a logical file name one can use a web browser, command line client or HTTP API to issue an HTTP request like

```
http://spitfire.cern.ch/getPFNs.xsql?table=repcat&lfn=lfn://cms.org/file1
```

Performance. With the design goal to provide high performance and low latency Java servlets were chosen since they are highly efficiency and provide a scalable solution [18]. To minimize the number of connection setups, persistent HTTP(S) 1.1 connections [14] are used (but not required). For similar reasons, the services use thread and JDBC connection pooling, and advanced caching. The net effect is that requests and responses are passed through layers with very little delay.

Security. The security mechanism in Spitfire 1.1.0 is performed in two steps, as outlined in figure 1: first a user is authenticated, and then he/she is authorised to perform the requested action.

In the first step, the authentication starts with verifying the validity of the client's certificate. The certificate is sent together with the HTTP request when the SSL connection is established. The verification is made in two phases: first the certificate's signature is verified against the set of trusted Certificate Authorities' certificates. Next, the certificate is checked against a Certificate Revocation List by the TrustManager from within the SSLServerSocketFactory class.

If the certificate passes both tests, the client is properly authenticated and the second step – the authorisation– starts: the request passes to the core servlet of Spitfire, Oracle XSQL Servlet (version 9.0.2.0.0C Beta). This servlet selects the XSQL file corresponding to the client's request. The file itself describes a data base operation. These files (db operations) can be protected by embedding an XML tag for a tailored action handler into the file. When the servlet encounters this tag, it invokes the class implementing the interface. The action handler interface itself is called XSQLActionHandler.

In Spitfire's case the chosen tag name is `<xsql:auth>` and it is used for authorising the client's request. When the servlet encounters this tag when parsing an XSQL file, it makes a call to the class AuthorizationXSQLHandler. This class checks the client's request for a role parameter and verifies against the role data base if the client is allowed to act in the given role. The verification is done using the certificate subject name of the client's X.509 certificate.

Finally, the role is mapped into a connection identifier which is understood by the Oracle XSQL servlet. The servlet uses this identifier to connect to the actual data base attached to the XSQL servlet for performing the requested operation.

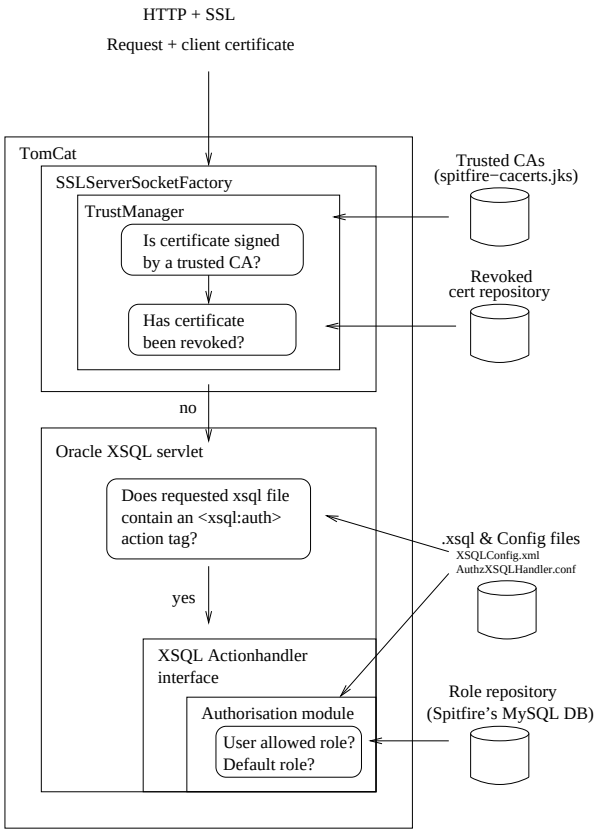
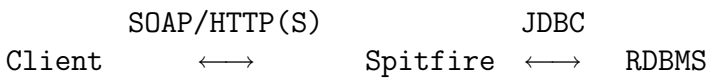


Fig. 1. Spitfire authentication and authorisation mechanisms

3 Spitfire Web Service Architecture

The *web services* vision [21, 11, 22] is a step towards increased Grid software execution allowing distributed computing where programs are no longer configured with static information. Within this vision programs are made more flexible and powerful by querying Internet databases (registries) at runtime in order to discover information and network attached third-party building blocks. Services advertise themselves and related metadata via such databases, enabling the assembly of distributed higher-level components. A web service consists of a set of interfaces with associated operations. Each operation may be bound to one or more network protocols and endpoints. The definition of interfaces, operations and bindings to network protocols and endpoints are given in a standard grammar such as the Web Service Description Language (WSDL) [19]. In contrast to popular belief, a web service is neither required to carry XML messages, nor to be bound to SOAP or the HTTP protocol, nor to run within a .NET [23] hosting environment (although all of these technologies may be helpful for implementations).

Work is in progress to evolve Spitfire towards a Grid web service (v1.2.0alpha). In the communication layer between client and Spitfire, XML/HTTP(S) is replaced with SOAP/HTTP(S), leading to an architecture that can be summarized as follows:



3.1 Spitfire as a Web Service

The Spitfire Web Service API is defined in section 5 and implemented using Remote Procedure Call (RPC) over SOAP. The Spitfire service is designed to sit in front of a database, either new, or a pre-existing one. It only requires database administrator (DBA) access once to set up the initial accounts, and the relevant table spaces to be managed by Spitfire.

3.2 Web Service Engine

The Spitfire service incorporates the Apache Axis web service engine [24], currently beta 2 version. The rationale behind this choice is that Axis is currently one of the most flexible web service engines available, and moreover it is open source. It also has a HTTP binding implemented as a servlet, and so can run side-by-side with our XSQL-based version of Spitfire on any servlet engine. The development was done on top of the Jakarta Tomcat v4.0.3 servlet engine [20]. The resulting service architecture is illustrated schematically in figure 2.

Axis provides handlers for both RPC based SOAP calls and Message based SOAP calls. Since a defined programming API needed to be implemented, the SOAP-RPC handler of Axis was best suited. Axis provides multiple bindings to different transports, with the additional possibility to write user-defined transport listeners. From the available transport listners, the

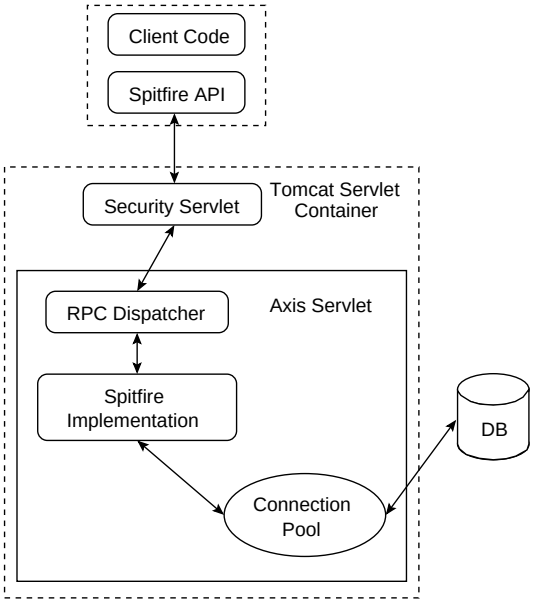


Fig. 2. Spitfire server as a web service

HTTP transport binding was chosen in order to keep the client code as simple as possible and to allow the server code to run in a servlet container.

The architecture of the Axis engine starts at the client side, where the RPC parameters are encoded in SOAP-RPC. The Axis library handles all the serialisation of standard Java classes. More specialised classes can be serialised and deserialised easily using a standard Axis mechanism which works with JavaBean components. Axis also provides hooks for writing customized serialisers, should the class not be written as a bean. In order to maximise cross-language compatibility, only standard data types, common across many languages, were chosen.

The HTTP binding of the Axis engine is implemented as a normal servlet. The HTTP call is dealt with by the HTTP transport listener of the Axis engine, which uses the call to determine which Axis request chain to invoke. An Axis request chain is a sequence of handlers that process an incoming SOAP message. We use the `RPCDispatcher` handler, with the relevant class for the RPC being determined by the URL of the HTTP call. No other handlers are defined to process the message.

The `RPCDispatcher` decodes the SOAP message, determining which method of the specified class should be called, and handles the deserialisation of the parameters so they can be passed to the correct method call. After the call has completed, the RPCDispatcher handles the serialisation of the return value (or of the `SOAPFault`, in case of an error) and hands this SOAP Message to the transport mechanism for return to the client.

On a more technical level, we note that the Spitfire service is defined to have *Application Scope* so that each new call is handled by the same instance of the RPC service class, in a manner very similar to the way in which servlets themselves operate. It is important, therefore,

to ensure that the implemented methods are correctly synchronised. The Application Scoping allows the service to know about requests from other calls, possibly permitting optimisations in the future. Currently, however, although the methods are synchronised, no such cross-call optimisation has been implemented.

3.3 Connection Pool

A thread running on the server maintains the connection pool to the database backend. The pool is organised by role, i.e. there is effectively a single pool for every defined role of the Spitfire service. Once a client has been authorised in a particular role, a connection is leased from the pool for the duration of the call. Pooled connections to the database remain open, even when they are not in use; this is to avoid the JDBC connection overhead to the backend database being imposed on every call. Only when the pool is expanded is there an overhead, and the pool is designed to handle this in an optimised fashion.

The API also defines that a connection may be leased for a longer period to permit the client to keep the connection to the server open for subsequent calls; this is more efficient since it avoids the TCP connection and security overhead. This functionality is yet to be implemented.

3.4 Translation

The Spitfire client API is designed for the SQL99 standard [25]. This standard is not implemented completely by most RDBMS, and often implemented with numerous variations. Additionally, exceptions and errors from the database backend are not standardised. A remote client should not have to know how to decode exceptions from the particular backend fronted by the service. The translator has two roles:

- To handle the translation of the client supplied call into an SQL statement appropriate for the particular database backend that Spitfire is using.
- To ensure uniform exceptions and errors are returned to the client regardless of the database backend type. This also serves to wrap errors so that the remote client only sees errors that are relevant for them, and that they are permitted to see by the local security policy.

4 Security Mechanisms

4.1 Security in the current Spitfire

The security module in the development version of Spitfire is quite different from the module in version 1.1.0, in fact the module is now an independent servlet, as outlined in Figure 3. This servlet performs the same function as the original authorisation module AuthorizationXSQLHandler in version 1.1.0 but it is independent of the Oracle XSQL servlet and its XSQLActionHandler interface.

After the initial authentication of the client, the request is now passed to a servlet chain of which the security servlet is the first one. If the client is successfully authorised then the

role is mapped to a connection identifier and the request is passed to the XSQL servlet. If not, an error message is returned.

The role verification step is now separate from the mapping of the role to a connection identifier. This provides for a two phase authorisation where the mapper module can be switched to another module so that the servlet may be adapted to protect other servlet based applications as well.

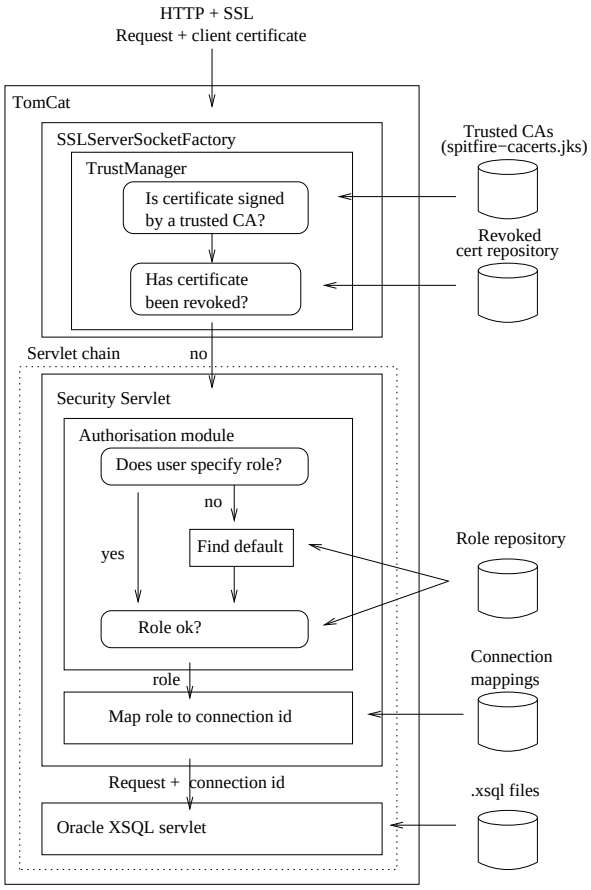


Fig. 3. Spitfire authentication and authorisation with a security servlet

4.2 Security in the Web service Spitfire

The final decision on which security solution to use for passing the role information used for end user authorisation in the web services based Spitfire is still to be made but there are three main options that are being evaluated. All of them have their benefits and drawbacks.

The cleanest and technically most advanced option is embedding role information inside the user's proxy certificate. Storing this information in a certificate extension would also

be backwards compatible with the rest of EDG software using mostly Globus style proxy certificates. In terms of implementation, this is however the most elaborate as software for both generating these special extensions in the client end and parsing them in the server end need to be implemented.

The second option is modifying the existing authorisation servlet of Spitfire to look for the user's role information in the SOAP message itself. Even though rather straightforward to implement, this induces a performance penalty as the SOAP message would need to be parsed twice: first by the authorisation servlet to verify the role and next by Axis itself performing the request.

The last option is writing a custom handler to the Axis request chain and if possible reusing code from the existing authorisation servlet. In terms of implementation effort, this falls somewhere in between the former two. In this case we still have the problem of how the the role parameter gets passed from client to server.

5 Application Programming Interface (API)

The client RPC API for the web services version of Spitfire is divided into five independent interfaces. Any specific client need only implement one of these to be termed a *Spitfire client*. The five interfaces are described briefly below. A fuller explanation of these APIs, including a definition of all the methods, is given in Appendix A.

Grid Database Administration. This API provides administrative functionality over the Grid-enabled databases, for example, permitting the Grid administrator to create and delete new databases in the Grid tablespace. Note that the privileges associated with this API are distinct from and less than the privileges of the real Database Administrator on the backend database; the Grid Admin API, for example, cannot create or delete non-Grid users of the DBMS, nor is it permitted to create databases or tables outside of the assigned Grid tablespace. The functions provided are `createDatabase`, `deleteDatabase`, `createTable`, `dropTable` and `alterTable`.

User Management. As described earlier, an authenticated user is assigned a role by the security mechanisms. Connection to the database is via this role, and the backend database is then responsible for enforcing the authorisation of that role. Consequently, the purpose of this API is both to define the role names and what they mean to the backend database (in terms of privileges), and also to assign users to one or more of these defined roles. The functions provided are `createRole`, `deleteRole`, `updateRole`, `registerUsers`, `unregisterUsers` and `updateUserRole`.

Database Information. This API permits clients to extract information about the Grid database, its associated tables and their schema. It also permits querying of the security roles and user to role mappings. The functions provided are `showTables`, `showTableColumns`, `showColumnType`, `showCreateTable`, `showDBSpace`, `showDBRoles` and `showRoleUsers`.

Core SQL Functionality. This defines methods implementing the common SQL operations for select, insert, update and delete. It also permits timestamping and the establishment of longer-term connections to the service. The functions provided are `executeSQL`, `select`, `insert`, `update`, `delete`, `openConnection`, `closeConnection`, `setTableTimeStamp`, `setRowTimeStamp`.

High-level Functionality. No higher level functionality is defined at the moment. Future work on Spitfire in the areas of replication, distribution and clean-up services, is discussed in section 7.

6 Example Deployment

This section gives a short example of how to implement a real API using Spitfire. It is divided into two parts. The first part briefly outlines usage of the XSQL version of Spitfire (v1.1.0), while the second part shows usage with the web services version of Spitfire (v1.2.0alpha). Further examples are given in Appendix B.

The example given is from the Work Package 2 Replica Metadata Catalogue (RepMec) whose responsibility is to keep track of the metadata pertinent to the Logical File Names (LFNs) of the Replica Management System. This information includes both application specific metadata, e.g. run number and physics information associated to the file's production, and also technical data for internal use by the Replica Management System, e.g. access control lists.

The example focuses on one of the tables defined for RepMec, namely the technical metadata table. The table is defined as follows:

Field	`LFN`	`origOwner`	`origACL`	`masterLRC`	size
Type	String	String	String	String	Integer

- `LFN` is the Logical Filename and is the primary key of the table.
- `origOwner` is an identifier for the user who created the master copy of this file on the Grid.
- `origACL` is the original Access Control List of the file as defined by the `originalOwner`.
- `masterLRC` is the URL reference for the Local Replica Catalogue (LRC) on which the master copy of this file is located.
- `size` is the file size in bytes.

Only one method is described here, a basic `SELECT` that finds the LRC that hosts the master copy of this LFN:

```
(String) master lookupMaster(LFN)
```

Further method implementations giving examples of `INSERT`, `UPDATE` and `DELETE` operations are presented in Appendix B. For further information about the RepMec API, refer to the Replication MetaData Database documentation [26]. In the example, we assume that the table, as described above, is contained within a database called 'GRID'. We also assume that the client is appropriately authorised on the Spitfire server.

6.1 XSQL based Spitfire

This section describes how the example is implemented using v1.1.0 of Spitfire, based on the use of server side XSQL template files. The process of making an operation upon the database requires the pre-definition of server side templates. The named parameters (e.g. {`@lfn`}) in these templates are substituted with the actual parameter attributes of the HTTP GET or POST request. The example method is implemented by specifying a database SELECT in the associated XSQL file:

```
<?xml version="1.0"?>
<page xmlns:xsql="urn:oracle-xsql">
<xsql:auth user_role = "{@role}" auth_policy="{@policy}"></xsql:auth>
<xsql:query xmlns:xsql="urn:oracle-xsql"
        null-indicator="yes"
        allow-client-style="yes"

        SELECT lfn, masterlrc FROM lfndata WHERE lfn={@lfn}
</xsql:query>
</page>
```

The `page` tag belongs to the XSQL schema. The `xsql:auth` tag is required to ensure that the client is authorised by the Spitfire security mechanisms. The client-side method implemented in RepMec simply sends the requested lookup LFN to this file's URL with the `lfn` parameter encoded in an HTTP GET request, e.g.

```
http://repmec001.cern.ch/repmec/lookupMaster.xsql?lfn=lfn://file1.cern.ch/dat/file1234
```

The HTTP request returns the result of this query as an XML rowset in canonical XML (see Section 2). The client-side method can parse the results and retrieve the `masterLRC` information requested by the client.

6.2 Web Services Based Spitfire

This section describes how the functionality of the example can be implemented using the Java web services API described in Section 5. A single call to the `select` client method is all that is required.

```
import org.edg.spitfire.client.SpitfireCoreServiceLocator;
import org.edg.spitfire.client.SpitfireCore;
  :
  :
String lookupMaster(String LFN)
       throws RepMecException {

   try {
      SpitfireCoreService service = new SpitfireCoreServiceLocator();
      SpitfireCore port = service.getSpitfireCore();

      // This makes the call to the service
      String resultSet = port.select("GRID",                          // database name
```

```
                                    new String[] { "lfn","masterlrc" }, // select attributes
                                    new String[] { "lfndata" ),         // tables
                                    "",                                 // no JOIN
                                    new String ("lfn=" + LFN)           // WHERE clause
                                    );
      // parse it.
      return RowsetParser.parseXML(resultSet, "masterlrc");
   } catch (SOAPFaultException error) {
      error.printStackTrace();
      throw new RepMecException("Select method failed");
   }
}
```

The select method returns an XML resultset as a string, which is then parsed to retrieve the `masterLRC` information requested by the client.

7 Future Work

The latest stable Spitfire release (v1.1.0) is currently under active use by the EDG application work packages on the experimental testbeds. Support for this will continue, although development will be frozen other than for bug fixes. Work now concentrates on evolving Spitfire towards a Grid web service (v1.2.0alpha) We are already benefiting considerably from interaction with the user community, and are taking into consideration their views and requirements in the next steps for the product.

The immediate plans are to improve the alpha quality of the Spitfire web services version, with a more complete implementation of the specified API, as described in Section 5. Tests will be made to determine the stability and performance of the service under heavy loads and under differing types of request patterns.

Currently the alpha service only officially supports the MySQL database backend, so effort will be put into supporting more backends, including PostgreSQL, Oracle and DB2. The Globus certificate based security architecture defined and implemented for the v1_1_0 version of Spitfire need to be integrated with the web services alpha code.

We are committed to the development of the Open Grid Services Architecture and Spitfire products within the Global Grid Forum and Data Access and Integration Services Working Group of the GGF, and welcome the opportunity to integrate further with developments of the GGF.

The medium term plans are to implement more advanced functionality to enhance the usefulness of the Spitfire meta-data service:

- Functionality will allow replication of databases or database views to other Spitfire Grid services. This should improve performance by increased data locality. It should also improve availability, scalability, and robustness.
- Distribution and load balancing services are necessary to limit the load on servers. Particularly important is the ability to distribute a single database over several servers and the transparent integration through distributed query processing.

– Clean-up functionality is required for the automatic removal of stale soft state information based on timestamps and expiry information. The current proposal is to have one timestamp on each row of data in the service. An API will be provided to external services, such that they can query the timestamp information and clean up the databases for which they are responsible, in any desired manner.

8 Conclusions

Many Data Grid services maintain persistent metadata in remote relational databases. However, existing relational database systems are neither grid enabled nor web service enabled, adversely affecting cross-organizational interoperability and reuse. The European Data Grid's Data Management Work Package addresses these issues with *Spitfire*. The Spitfire service grid-enables a wide range of relational database systems by introducing a uniform service interface, data model, network protocol and security model. These are based on widely accepted standards and neutral with respect to programming language, platform and database product.

The current stable release places grid middleware between client and RDBMS, communicating relational data from a canonical XML data model over GSI-enabled HTTP(S). The service interface uses URL encoding of function names and parameters. Authorisation is role based. JDBC is used to interface to a wide range of existing backend RDBMSs.

The Spitfire Web Service is an evolution of that, and work in progress. It substitutes SOAP for client/server communication. The client API provides functionality for core SQL processing, database administration, user management and database information. Spitfire performs transparent translation of the input query and output result set to and from the target database backend. This is done in a secure, efficient and flexible manner, based on open standards.

9 Acknowledgements

We wish to thank Javier Jaen-Martinez for many fruitful discussions.

References

1. Ian Foster, Carl Kesselman, and Steve Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int'l. Journal of Supercomputer Applications*, 15(3), 2001.
2. Large Hadron Collider Committee. Report of the LHC Computing Review. Technical report, CERN/LHCC/2001-004, April 2001. http://lhc-computing-review-public.web.cern.ch/lhc-computing-review-public/Public/Report_final.PDF.
3. German Cancio, Steve M. Fisher, Tim Folkes, Francesco Giacomini, Wolfgang Hoschek, Dave Kelsey, and Brian L. Tierney. The DataGrid Architecture. Technical report, DataGrid-ATF-01, July 2001.
4. Wolfgang Hoschek, Javier Jaen-Martinez, Asad Samar, Heinz Stockinger, and Kurt Stockinger. Data Management in an International Data Grid Project. In *1st IEEE/ACM Int'l. Workshop on Grid Computing (Grid'2000)*, Bangalore, India, December 2000.
5. Wolfgang Hoschek, Javier Jaen-Martinez, Peter Kunszt, Ben Segal, Heinz Stockinger, Kurt Stockinger, and Brian Tierney. Data Management Architecture Report. Design, Requirements and Evaluation Criteria. Technical report, DataGrid-02-D2.2, September 2001.

6. Ann Chervenak, Ewa Deelman, Ian Foster, Wolfgang Hoschek, Adriana Iamnitchi, Carl Kesselman, Peter Kunszt, Matei Ripeanu, Heinz Stockinger Kurt Stockinger, and Brian Tierney. Giggle: A Framework for Constructing Scalable Replica Location Services. In *Proc. of the Int'l. IEEE Supercomputing Conference (SC 2002) (to appear)*, Baltimore, USA, November 2002.
7. Wolfgang Hoschek. A Database for Dynamic Distributed Content and its Application for Service and Resource Discovery. In *Int'l. IEEE Symposium on Parallel and Distributed Computing (ISPDC 2002) (to appear)*, Iasi, Romania, July 2002.
8. Francesco Giacomini. Definition of architecture, technical plan and evaluation criteria for scheduling, resource management, security and job description. Technical report, DataGrid-01-D1.2, September 2001.
9. Jason Lee, Dan Gunter, Martin Stoufer, and Brian Tierney. Monitoring Data Archives for Grid Environments. In *Proc. of the Int'l. IEEE Supercomputing Conference (SC 2002) (to appear)*, Baltimore, USA, November 2002.
10. Donald Bales. *Java Programming with Oracle JDBC*. O'Reilly, December 2001. ISBN 0-596-00088-x.
11. Ian Foster, Carl Kesselman, Jeffrey Nick, and Steve Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, January 2002.
12. Wolfgang Hoschek. The Web Service Discovery Architecture. In *Proc. of the Int'l. IEEE Supercomputing Conference (SC 2002) (to appear)*, Baltimore, USA, November 2002.
13. T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. *IETF RFC 2396*.
14. R. Fielding, J. Gettys, J.C. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. *IETF RFC 2616*. UC Irvine, Digital Equipment Corporation, MIT.
15. T. Dierks and C. Allen. The TLS Protocol Version 1.0. *IETF RFC 2246*, January 1999.
16. R. Butler, D. Engert, I. Foster, C. Kesselman, S. Tuecke, J. Volmer, and V. Welch. A National-Scale Authentication Infrastructure. *IEEE Computer*, 33(12), 2000.
17. World Wide Web Consortium. Extensible Markup Language (XML) 1.0. *W3C Recommendation*, October 2000.
18. World Wide Web Consortium. Simple Object Access Protocol (SOAP) 1.1. *W3C Note 8*, 2000.
19. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. *W3C Note 15*, 2001. www.w3.org/TR/wsdl.
20. Apache Software Foundation. The Jakarta Tomcat Project. http://jakarta.apache.org/tomcat/.
21. Wolfgang Hoschek. *A Unified Peer-to-Peer Database Framework for XQueries over Dynamic Distributed Content and its Application for Scalable Service Discovery*. PhD Thesis, Technical University of Vienna, March 2002.
22. P. Cauldwell, R. Chawla, Vivek Chopra, Gary Damschen, Chris Dix, Tony Hong, Francis Norton, Uche Ogbuji, Glenn Olander, Mark A. Richman, Kristy Saunders, and Zoran Zaev. *Professional XML Web Services*. Wrox Press, 2001.
23. Oracle. J2EE and Microsoft .NET, April 2002. Oracle Corp., White Paper.
24. Apache Software Foundation. The Axis Project. http://xml.apache.org/axis.
25. International Organization for Standardization (ISO). Information Technology-Database Language SQL. *Standard No. ISO/IEC 9075:1999*, 1999.
26. Leanne Guy, Peter Kunszt, Erwin Laure, Heinz Stockinger, and Kurt Stockinger. Replica Management in Data Grids. Technical report, Global Grid Forum Informational Document, GGF5, Edinburgh, Scotland, July 2002.

10 Appendix A: Complete Application Programming Interface

This Appendix describes more fully the RPC client API for the webservices version of Spitfire. As discussed in section 5, the API is divided into 5 independent parts.

- Grid Database Administration
- User Management
- Database Information
- Core SQL Functionality
- High-level Functionality

Each is discussed in more detail in the sections below, with a full description of all the defined methods.

10.1 Grid Database Administration API

This API provides administrative functionality over the Grid-enabled databases, for example, permitting the Grid administrator to create and delete new databases. Note that the privileges associated with this API are distinct from and less that the real Database Administrator on the backend database; the Grid Admin API, for example, cannot create or delete non-Grid users of the DBMS, nor is it permitted to create databases or tables outside of the assigned Grid tablespace.

createDatabase `error createDatabase(databaseName)`

The function creates a database within the Spitfire service. If there is not enough space on the database node for a database to be created an error is returned.
`databaseName:` the database name to be created.
`error:` an object containing a string describing the error and an associated error code[1].

deleteDatabase `error deleteDatabase(databaseName)`

The function deletes the specified database.
`databaseName:` the database name to be deleted.
`error:` an object containing a string describing the error and an associated error code.

createTable `error createTable(tableIdent, databaseName)`

[1] Here, any possible error return will be represented by the name 'error' as a return value. The implementation of how an error is handled is client language dependent. The error in Java, for example, is signalled to the client by throwing a `SOAPFault Exception`.

If the projected size of the table exceeds the disk space on the node running the backend, the command will fail and an error will be reported. The table schema is tested by creating the table and listing the contents. If the schema is bad then an error is returned.
`tableIdent:` a description of the table to be created.
`databaseName:` the name for the database, containing the table to be created.
`error:` an object containing a string describing the error and an associated error code.

dropTable `error dropTable(tableIdent,databaseName)`

`tableIdent:` a table identifier of the table to be dropped.
`databaseName:` the name for the database, containing the table to be dropped.
`error:` an object containing a string describing the error and an associated error code.

alterTable `error alterTable(tableIdent,databaseName, newTableIdent)`

`tableIdent:` a table identifier of the table to be altered.
`databaseName:` the name for the database, containing the table to be altered.
`newTableIdent:` the new description of the table.
`error:` an object containing a string describing the error and an associated error code.

10.2 User Management API

A described earlier, an authenticated user is assigned a role by the security mechanisms. Connection to the database is via this role, and the backend database is then responsible for enforcing the authorisation of that role. Consequently, the purpose of this API is both to define the role names and what they mean to the backend database (in terms of privileges), and also to assign users to one or more of these defined roles. The User Management API should be independent of the implementation of the security mechanisms.

createRole `error createRole(roleName,databaseName)`

This method creates a new role upon the specified database. The default, minimum database permissions are assigned to the role when it is created. To modify the permissions, the method `updateRole` should be called.
`roleName:` the name of the role to be created on the given database.
`databaseName:` the name of the database to which this role pertains.
`error:` an object containing a string describing the error and an associated error code.

deleteRole `error deleteRole(roleName,databaseName)`

This method deletes an existing role for the specified database.
`roleName:` the name of the role to be deleted.
`databaseName:` the database for which this role should be deleted.
`error:` an object containing a string describing the error and an associated error code.

updateRole `error updateRole(roleName,rolePermissions,databaseName)`

This method updates the privileges assigned to the given role on specified database. The exact implementation of the permission levels has not yet been finalised, but will correspond closely to those defined for SQL99 roles.
`roleName:` the name of the role to be updated.
`rolePermissions:` A string defining the permissions on the database for the given role.
`databaseName:` the name of the database whose role is being updated.
`error:` an object containing a string describing the error and an associated error code.

registerUsers `error registerUsers(userList,databaseName)`

This method registers users for access to the specified database. This method assigns only the default role to the user, which allows only minimal privileges. To assign a different role, i.e extra privileges, the method `updateUserRole` must be invoked. The userList may be LDAP Distinguished Names from Grid certificates for example.
`userList:` an array of user names to add to the access list on the given database.
`databaseName:` the database to which the users are to be allowed access.
`error:` an object containing a string describing the error and an associated error code.

unregisterUsers `error unregisterUsers(userList,databaseName)`

This method unregisters users for access to the specified database.
`userList:` An array of user names to be deleted from the access list of the given database.
`databaseName:` the database from which the users are to be deleted.
`error:` an object containing a string describing the error and an associated error code.

updateUserRole `error updateUserRole(userName,roleName,databaseName)`

This method modifies the role assigned to user on the given database.
`userName:` The user name for which the assigned role is to modified for the given database.
`databaseName:` the database in which the user-role combination is valid.
`error:` an object containing a string describing the error and an associated error code.

10.3 Database Information API

This API permits users to extract information about the Grid database, its associated tables and their schema. It also permits querying of the security roles and user to role mappings.

showTables `error, tables[] showTables(databaseName)`

This method simply returns the list of tables defined in the database specified by databaseName.
`databaseName:` the database name from which the tables should be listed.
`tables:` a list of strings, containing the table names defined in the database.
`error:` an object containing a string describing the error and an associated error code.

showTableColumns `error, columns[] showTableColumns(databaseName,tableName)`

This method simply returns the column names of a table defined in the database associated with databaseName.
`databaseName:` the database name from which the tables should be listed
`tableName:` a simple string containing the table name to be described.
`columns:` a list of column names.
`error:` an object containing a string describing the error and an associated error code.

showColumnType

```
error, columnType showTableColumns( databaseName,
                                    tableName,
                                    columnName  )
```

This method simply returns the column type of a given column/table.
`databaseName:` the database name.
`tableName:` a simple string containing the table name to be described.
`columnName:` a simple string containing the column name to be described.
`columnType:` a string type identifier, based on the SQL99 standard.
`error:` an object containing a string describing the error and an associated error code.

showCreateTable `error, sqlString showCreateTable(databaseName,tableName)`

This method returns the SQL statement that was used to create the table, thus fully describing it.
`databaseName:` a string containing the database and table from which the schema is extracted.
`tableName:` a simple string containing the table name to be described.
`sqlString:` an SQL99 CREATE TABLE string that was used to create the given table.
`error:` an object containing a string describing the error and an associated error code.

showDBSpace `error, total, remaining showDBSpace(databaseName)`

The total and remaining storage size is returned in KBytes.
`databaseName:` a string containing the database name.
`total:` the total amount of storage that is allocated to the database.
`remaining:` the remaining amount of storage that is available to the database users.
`error:` an object containing a string describing the error and an associated error code.

showDBRoles `error, roleList[] showDBRoles(databaseName)`

This method returns the roles known to the database with the associated permissions.
`databaseName:` a string containing the database for which a user summary is required.
`roleList:` is an array of role names and permission level, returned for a given databaseName.
`error:` an object containing a string describing the error and an associated error code.

showRoleUsers `error, userList[] showRoleUsers(databaseName, roleName)`

This method returns the users assigned to a given role.
`databaseName:` a string containing the database for which a role summary is required.
`roleName:` a string containing the role name.
`userList:` is an array of user names assigned to a given role.
`error:` an object containing a string describing the error and an associated error code.

10.4 Core SQL Functionality API

This API defines the common SQL operations performed upon existing database tables. Many of the methods are overloaded to avoid the syntax becoming too clumsy.

executeSQL `error, resultSet executeSQL(databaseName, rawSQL)`

Execute the SQL statement and return an XML resultset to the client containing the result of the operation. If the operation fails, an error is returned to the client. The operation is performed with whatever authorised role the user has upon the database; attempting an unauthorised operation will result in an error.
`databaseName:` a string specifying the database name.
`rawSQL:` the SQL statement to execute.
`resultSet:` an XML formatted resultset containing the result of the operation.
`error` is an object containing a string describing the error and an associated error code.

insert There are multiple overloaded versions of this function reflecting the varying ways in which a user can insert data. The most basic is given below:

```
error, rowsInserted insert( dataBaseName,
                            table,
                            attributes[],
                            values[]  )
```

```
error, rowsInserted insert( dataBaseName,
                            table,
                            HashTable{attributes, values} )
```

`databaseName:` a string specifying the database name.
`table:` the table name into which the insert will be made.
`attributes[]:` the ordered attributes of the table in which to insert, as an array. `values[]:` the ordered values to insert. An overloaded version of this function allows a attribute-value hashtable to be used instead.
`error:` an object containing a string describing the error and an associated error code.
`rowsInserted:` will be 1 in the case of a successful insert. This function inserts a single row into an SQL table. A second type of insert where more than one row is inserted is provided:

```
error, rowsInserted insert( dataBaseName,
                            table,
                            attributes[],
                            values[][] )
```

```
error,
rowsInserted insert( dataBaseName,
                     table,
                     MVHashTable{attributes, values} )
```

`attributes[]` is as before, but `values[][]` is now a double array, with the second element indexing the rows to be inserted. Support for a single-key multi-value hashtable is also provided.

`rowsInserted:` corresponds to the number of rows successfully inserted into the table. The final method inserts an entire result set. The resultset is XML based, and is of the same schema that is returned by a `s`elect query.

```
error, rowsInserted insert(dataBaseName, table, XML)
```

where the `XML` is either a string, or a XML DOM document object; it is currently the client's responsibility to validate the XML document against the XML schema and to ensure that the inserted data corresponds to the table into which it is being inserted. Should the XML document be malformed or the data unsuitable for the table, the error will be returned to the client.

The following should be noted when using the `insert` method.

- The values (whether passed as arrays or hashtables) should be generic Object class or strings. The meta-data service will handle the internal casting (and construct the appropriate insert string); in the case where there is a type mismatch, the database exception will be returned to the client. The client may find out the relevant type for a table by using the database querying API.
- The timestamp attribute of the row is set to the time it was inserted.
- The INSERT...SELECT method of SQL99 is supported in two stages by making the relevant SELECT and then inserting the returned result set.
- The DEFAULT option is not implemented explicitly. Missing out a column name from the `attributes[]` array will cause the default to be inserted, and generate an exception if no default is defined on the column.

update This method is overloaded, with two possible forms:

```
error, numberUpdated update( dataBaseName,
                             table,
                             values[],
                             attributes[],
                             WHERE  )
```

```
error, numberUpdated update( dataBaseName,
                             table,
                             Hashtable{values,attributes},
                             WHERE )
```

This function updates the specified fields of any matched rows. Fields which are not specified are left as they are.
`table:` is the table name into which the update will be made.
`attributes[]:` are the ordered attributes of the table in which to update, as an array.
`values[]:` are the ordered values to update. An overloaded version of this function allows an attribute-value hashtable to be used instead. The timestamp attribute of the row is updated.

`WHERE:` is the logical WHERE clause containing the rows to be matched, as described in the `select` method.
`error:` is an object containing a string describing the error and an associated error code.
`rowsUpdated:` will contain the number of matched rows which were updated; rows that are matched by the WHERE clause but not changed (i.e. the new attributes are the same as before) are not included.

select This method is overloaded, with two possible forms:

```
error, resultSet select( dataBaseName,
                         selectlist[],
                         tables[],
                         JOIN,
                         WHERE)
```

```
error, resultSet select( dataBaseName,
                         selectlist[],
                         tables[],
                         jointype,
                         JOIN,
                         WHERE)
```

This function offers basic select functionality.
`selectlist[]:` is an array of selected attributes to return in the resultset. If an element of the `selectlist[]` contains an '*' then all attributes of the table are output.
`tables:` is an array of table names from which to make the select.
`jointype:` is the type of join to be performed, either INNER, LEFT, or RIGHT. If missed out, the default is INNER JOIN.
`JOIN:` is the JOIN clause, discussed below.
`WHERE:` is the logical WHERE clause containing the rows to be matched, discussed below.
`error:` is an object containing a string describing the error and an associated error code.
`resultSet:` is an object containing the returned resultSet in XML format.

Current limitations. The current implementation does not support the GROUP BY, ORDER BY or HAVING clauses. The AS clause in the selectlist and tablelist is not supported; the FROM, JOIN and WHERE clauses use the full column name. Currently no aggregate functions are supported.

JOIN clause. Joins are made using an ANSI SQL99 JOIN clause; Oracle theta joins are not directly supported, although still possible.

- The default join is an inner join, with the unmatched rows of either table being discarded. The jointype, if specified, is 'INNER'.
- A left join, where unmatched rows on the left table are matched with a null from the right hand table, must be specified with jointype 'LEFT'.

– A right join, where unmatched rows on the right table are matched with a null from the left hand table, must be specified with jointype 'RIGHT'.

The ANSI CROSS JOIN operator is currently not supported since this can result in huge resultsets and is not generally recommended; this can currently be accomplished by specifying both tables in the tablelist and leaving the WHERE clause empty (this is a theta cross join). FULL joins are not supported. Currently multi-table (>2) joins are not supported.

WHERE clause. The WHERE clause is ANSI SQL99 expression. Theta joins (by prefixing the tablename) are not explicitly supported, although they may work, depending on the backend. Sub-selects are not explicitly supported.

delete This method is overloaded, with two possible forms:

```
error, numberDeleted delete(dataBaseName, table, WHERE)
```

```
error, numberDeleted delete(dataBaseName, table)
```

The second version of this is distinct from the `dropTable` method in the Admin API which deletes the table definition as well.
This function deletes any rows in the specified table matched by the `WHERE` clause.
`table` is the table name on which the `delete` will operate.
`WHERE:` is the logical `WHERE` clause containing the rows to be matched, as described in the `select` method. If the `WHERE` clause is absent, all rows in the table are deleted.
`error:` is an object containing a string describing the error and an associated error code.
`rowsDeleted:` will contain the number of rows which were deleted.

setTableTimeStamp

```
error setTableTimeStamp(databaseName, table, timeStamp)
```

Updates the timestamp on a table.
`table:` the table whose timestamp is to be updated.
`timeStamp:` the recommended Grid Forum timestamp.

setRowTimeStamp

```
error, setTableTimeStamp( dataBaseName,
                          table,
                          WHERE,
                          timeStamp )
```

Updates the individual row timestamps matched by the `WHERE` clause.
`table:` the table whose rows are to be timestamped.
`WHERE:` the logical WHERE clause containing the rows to be matched.
`timeStamp:` the recommended Grid Forum timestamp.

openConnection

```
error, lease, connection openConnection( dataBaseName,
                                         lease )
```

The purpose of this method is to open a connection to the database servlet and maintain the connection open. This has advantages if you wish to make a number of database operations over a longer period of time, and do not want the HTTP(S) connection overhead every time you execute a new database command.
`dataBaseName:` the name of database with which to connect.
`lease:` is the requested idle timeout in seconds before the connection is dropped. The returned `lease` is the actual timeout given to you by the database servlet (which may be less than requested).
`connection:` the an object representing the connection to the database. It may be used in all methods in place of the dataBaseName parameter, to indicate that this open connection should be used rather than making a new one.
`error:` an object containing a string describing the error and an associated error code.

closeConnection

```
error closeConnection(connection)
```

Close the associated connection.
`connection:` the object corresponding to the connection to be closed.
`error:` an object containing a string describing the error and an associated error code.

11 Appendix B: Expanded Example Deployment

This appendix provides more complete examples of how to implement a real API. For completeness, some material given in section 6 is repeated here. As in section 6, the description is split into two parts. The first part briefly outlines the mechanism to for usage of the XSQL version of Spitfire (v1.1.0), while the second part shows usage of the web services version of Spitfire.

The example given is from the Work Package 2 Replica Metadata Catalogue (RepMec) whose responsibility is to keep track of the metadata pertinent to the Logical File Names (LFNs) of the Replica Management System. This information includes both application specific metadata, e.g. run number and physics information associated to the file's production, and also technical data for internal use by the Replica Management System, e.g. access control lists.

11.1 RepMec Example Table

The example focuses on one of the tables defined for RepMec, namely the technical meta-data table. The table is defined as follows:

Field	`LFN`	`origOwner`	`origACL`	`masterLRC`	size
Type	String	String	String	String	Integer

- `LFN` is the Logical Filename and is the primary key of the table.
- `origOwner` is an identifier for the user who created the master copy of this file on the Grid.
- `origACL` is the original Access Control List of the file as defined by the `originalOwner`.
- `masterLRC` is the URL reference for the Local Replica Catalogue on which the master copy of this file is located.
- `size` is the file size in bytes.

Some of the client APIs pertinent to this table are given below. For brevity, much of the API is omitted here; we only examine four methods.

- `(void) insertLFN(LFN, owner, acl, lrc, size)`
  The basic method to add another entry to the RepMec table.
- `(String) master lookupMaster(LFN)`
  To find the LRC that hosts the master copy of this LFN.
- `(void) removeLFN(LFN)`
  To delete the entry for this LFN. Called by the replica manager while deleting a LFN from the Grid.
- `(void) updateMaster(LFN, newMasterLRC)`
  To update which LRC hosts the master of this LFN. Called by the replica manager while moving the master flag on a LFN.

Here, the discussion is limited to implementing this table and the above methods. Refer to the Replication MetaData Database documentation [26] for further information about the design and architecture of the RepMec system itself.

11.2 Preparing the Database Backend

The first part is to create the database and table on the relevant backend. Although this operation can be accomplished using a template, it is probably easier, provided one has access, to create the database manually upon the backend, since this operation need only be done once. Here, it is assumed that the reader is familiar with SQL and with the specific backend database implementation of their choice, therefore the details of table creation are simply summarised as a CREATE TABLE statement. We assume that all operations are made on a database called 'GRID'.

```
CREATE TABLE lfndata ( LFN varchar(250)
                       ORIGOWNER varchar(128)
                       ORIGACL varchar(128)
                       MASTERLRC varchar(250)
                       SIZE int(11)
                       PRIMARY KEY (LFN) )
```

11.3 XSQL based Spitfire (v1.1.0)

This section describes how the example is implemented using v1.1.0 of Spitfire, based on the use of server side XSQL template files. The process of making an operation upon the database requires the pre-definition of server side templates. The named parameters (e.g. {`@lfn`}) in these templates are substituted with the actual parameter attributes of the HTTP GET or POST request. The example method is implemented by specifying a database SELECT in the associated XSQL file:

```
<?xml version="1.0"?>
<page xmlns:xsql="urn:oracle-xsql">
:
:
</page>
```

All pages here are also required to be authorised by the security mechanisms, therefore, an additional `xsql:auth` tag is included:

```
<?xml version="1.0"?>
<page xmlns:xsql="urn:oracle-xsql">
<xsql:auth user_role = "{@role}" auth_policy="{@policy}"></xsql:auth>
:
:
</page>
```

This ensures that the default role is chosen for a client based on the certificate DN, or if a role is specified, that the client is permitted to use that role. The following assumes that the client is appropriately authorised on the Spitfire service.

insertLFN `void insertLFN(LFN, owner, acl, lrc, size)`

The insert API can be implemented in two ways.

1. Basic Data Manipulation Language (DML) tag.
   A DML tag is suitable for one-shot inserts and is the most likely mechanism. It has the advantage that the client need only specify the insert values in the HTTP GET or POST request. Here, a basic DML tag is used with the parameters on the SQL INSERT statement:

```
<?xml version="1.0"?>
<page xmlns:xsql="urn:oracle-xsql">
<xsql:auth user_role = "{@role}" auth_policy="{@policy}"></xsql:auth>
<xsql:dml xmlns:xsql="urn:oracle-xsql"
        null-indicator="yes"
        allow-client-style="yes"
        table        = "lfndata">

        INSERT INTO lfndata
           VALUES ('{@lfn}', '{@owner}', '{@acl}', '{@lrc}', {@size})
</xsql:dml>
</page>
```

   On the server side, this file should be saved to, e.g. `insertOne.xsql` and placed in a path within the servlet engine so that it is accessible to external web clients. Throughout, it is assumed that the reader is familiar with the deployment and configuration of the Tomcat servlet engine. The client then explicitly GETs or POSTs the parameters (`lfn`, `owner`, `acl`, `lrc`, `size`) to the URL of the XSQL file, as URL encoded name-value pairs.
   If the client interface is an interactive page, the easiest way to make this post is via a web form, where the submit target for the form is set to the URL for the `insertOne.xsql` file. Automated clients simply need to use one of the many available HTTP libraries to construct a suitable POST or GET request. The RepMec software contains an implementation of the RepMec client API which hides the HTTP interactions from the client.
2. XSQL Insert tag.
   This is appropriate for bulk insert of LFNs. The disadvantage is that the client must construct and send a rowset XML string to use this method. The file `insertMany.xsql` is given below:

```
<?xml version="1.0"?>
<page xmlns:xsql="urn:oracle-xsql">
<xsql:auth user_role = "{@role}" auth_policy="{@policy}"></xsql:auth>
<xsql:insert-request table="lfndata">
</xsql:insert-request>
</page>
```

   The posted XML must be of the rowset form returned by the a SELECT query, an example is shown below. Since the input format for an insert is the same as the output format from a select query, this makes this method suitable for the large scale export and re-import operation required if copying the whole database.

```
<rowset>
 <row>
  <lfn>lfn://mcfile.cern.ch/dataset55/file1234</lfn>
```

```
    <origowner>mcgenerator</origowner>
    <origacl>mcgenerator:all,reconstruction:read</origacl>
    <masterlrc>lrc://filestore1.cat.rl.ac.uk</masterlrc>
    <size>123456789</size>
   <row>
   <row>
    <lfn>lfn://mcfile.cern.ch/dataset55/file8899</lfn>
    <origowner>mcgenerator</origowner>
    <origacl>mcgenerator:all,reconstruction:read</origacl>
    <masterlrc>lrc://filestore1.cat.rl.ac.uk</masterlrc>
    <size>2345678901</size>
   <row>
  </rowset>
```

lookupMaster `String master lookupMaster(LFN)`

The lookup methods keyed on the LFN are all implemented in the same way, by using a database SELECT in the associated XSQL file.

```
<?xml version="1.0"?>
<page xmlns:xsql="urn:oracle-xsql">
<xsql:auth user_role = "{@role}" auth_policy="{@policy}"></xsql:auth>
<xsql:query xmlns:xsql="urn:oracle-xsql"
        null-indicator="yes"
        allow-client-style="yes"

        SELECT lfn, masterlrc FROM lfndata WHERE lfn={@lfn}
</xsql:query>
</page>
```

The client method implemented in RepMec simply sends the requested lookup LFN to this file's URL with the `lfn` parameter encoded in an HTTP GET request, e.g.

```
http://repmec001.cern.ch/repmec/lookupMaster.xsql?lfn=lfn://file1.cern.ch/dat/file1234
```

The HTTP call returns the result of this query as a rowset, in the format shown above, whereupon the method decodes the XML rowset and returns the correct result to the client.

removeLFN `void removeLFN(LFN)`

This method, for the deletion of the specified LFN data from the catalogue, is again implemented using an XSQL DML tag:

```
<?xml version="1.0"?>
<page xmlns:xsql="urn:oracle-xsql">
<xsql:auth user_role = "{@role}" auth_policy="{@policy}"></xsql:auth>
<xsql:dml xmlns:xsql="urn:oracle-xsql"
        null-indicator="yes"
        allow-client-style="yes"
        table        = "lfndata">
```

```
        DELETE FROM lfndata WHERE lfn={@lfn}
</xsql:dml>
</page>
```

Again, the client method implemented in RepMec simply sends the requested LFN to this method as an HTTP parameter in a GET or POST request, e.g.

```
http://repmec001.cern.ch/repmec/removeLFN.xsql?lfn=lfn://file1.cern.ch/dat/file1234
```

The HTTP call returns a message indicating the number of rows deleted. If this is incorrect, the method raises an error.

updateMaster `void updateMaster(LFN, newMasterLRC)`

This method, used to change the location of the master copy of a file in the catalogue, is again implemented using a DML tag. Only the SQL is given below:

```
UPDATE lfndata WHERE lfn={@lfn} SET masterlrc={@newmasterlrc}
```

The number of rows updated is returned from the HTTP call and used to determine the success or failure of the method call.

11.4 Web Services Based Spitfire

This section describes how the functionality of the API can be implemented using the web services API described in Section 5. Again, the following assumes that the client is appropriately authorised on the Spitfire service.

insertLFN `void insertLFN(LFN, owner, acl, lrc, size)`

A single call to the `insert` client method is all that is required. Either a hashtable or two ordered arrays must be used to encode the column name-data pairs, with the column name being the key of the hashtable. An example code fragment for the Java API is given below:

```
import org.edg.spitfire.client.SpitfireCoreServiceLocator;
import org.edg.spitfire.client.SpitfireCore;
  :
  :
void insertLFN(String LFN, String owner, String acl, String lrc, int size)
       throws RepMecException {
   Hashtable data = new Hashtable();
   data.add("lfn", LFN);
   data.add("origOwner", owner);
   data.add("origACL", acl);
   data.add("masterLRC", lrc);
   data.add("size", size.toString());
```

```
   try {
      SpitfireCoreService service = new SpitfireCoreServiceLocator();
      SpitfireCore port = service.getSpitfireCore();

      // This makes the call to the service
      int rowsInserted = port.insert("GRID", "lfndata", data);

   } catch (SOAPFaultException error) {
      error.printStackTrace();
      throw new RepMecException("Insert method failed");
   }
}
```

Multiple entries can be entered using a multivalued hashtable or a two dimensional `value[][]` array. Note that in the case of Java, an error from the service is signalled as a `SOAPFaultException` to the client code. In the RepMec implementation, this error is caught and a more general RepMec error is thrown to the RepMec client.

lookupMaster `String master lookupMaster(LFN)`

Implemented in a similar manner to the previous example except that the `select` method is used. The extra code is omitted.

```
String resultSet = port.select("GRID",
                               new String[] { "lfn","masterlrc" },
                               new String[] { "lfndata" ),
                               "",     // no JOIN
                               new String ("lfn=" + LFN)
                               );
```

The method can then parse the returned XML resultset and extract the information to be returned to the client (in this case the information is the name of the master LRC for the specified LFN).

removeLFN `void removeLFN(LFN)`

This is implemented with a call to the `delete` method:

```
int numDel = port.delete( "GRID", "lfndata", new String ("lfn=" + LFN) );
```

If the `delete` method throws a `SOAPFaultException` or if the `numDel` is equal to zero, then the `removeLFN` method has failed and the client is informed.

updateMaster `void updateMaster(LFN, newMasterLRC)`

This function is implemented with a call to the `update` method. As with `select`, this involves either a Hashtable or two arrays to specify exactly which of the columns should be

updated with new values. It also requires a `WHERE` clause to determine which rows should be updated in the table. Here, the array mechanism is shown:

```
String [] attributes = { "masterlrc" };
String [] values = { new String(newMasterLRC) };
int numUpd = port.update( "GRID",
                          "lfndata",
                          values,
                          attributes,
                          new String("lfn=" + LFN)
                          );
```

If the `update` method throws a `SOAPFaultException` or if the `numUpd` is equal to zero, then the `updateMaster` method has failed and the client is informed.